Guidelines for the development of a communication middleware for automotive applications

Ricardo Santos Marques, Françoise Simonot-Lion
LORIA - INPL
Campus Scientifique, BP 239
54506 Vandoeuvre-lès-Nancy - France
{santos,simonot}@loria.fr
tel: +33 3 83 58 17 28, fax: +33 3 83 58 17 01

Abstract

An automotive middleware layer masks the heterogeneity of platforms, and provides high level communication services to applicative tasks. In addition, it is a software architecture, shared between car makers and third-part suppliers, ensuring the portability and interoperability of the applicative tasks.

In this study, a method aiming at developing the middleware's software architecture, and obtaining feasible scheduling parameters for network frames and middleware and applicative tasks, is presented. The architecture is built with a set of design patterns, and identifies a set of tasks executing the middleware's communication services. The scheduling parameters of frames and tasks are determined such that the timing constraints on tasks and signals are met.

1. Introduction

Automotive embedded systems are composed of a set of nodes, called Electronic Control Units (ECUs), interconnected by communication networks. On each ECU, applicative tasks execute periodically control algorithms, and most usually, are constrained by a relative deadline that is the maximum time interval tolerated between the activation of an instance of the task, and its completion. Moreover, automotive functions may be performed by several distributed applicative tasks, and thus, these tasks communicate by producing and consuming signals (e.g. the number of RPM of the engine) that are sent over the networks. These signal exchanges are also constrained by deadlines that limit the time interval between the consumption of a signal value on an ECU, and the activation of the instance of the remote task that produced that value.

In this context, the goal of a middleware layer is, on the one hand, to mask on each ECU the heterogeneity of communication platforms, and, on the other hand, to offer specialized services such as I/O abstraction or communication mechanisms independent of applicative tasks location. In this study, the emphasis is given on the following set of communication services: sending of produced signals, and reception of signals to be consumed. Since car makers purchase components developed by third-part suppliers, this middleware layer becomes a software architecture, shared between these actors, which must ensure the portability and the interoperability of the applicative level code. Moreover, the execution of the middleware's communication services interferes with the applicative tasks running on an ECU, and hence, increases the probability of the timing constraints associated to tasks and signals not being met.

Our goal is to propose a method, presented in figure 1, aiming at automatizing the development of the middleware's software architecture, and the setting of feasible scheduling parameters on the network frames and on the tasks. The middleware's software architecture is developed in order to improve the maintenance and the reusability of the software components. This way, it is easily exchanged between car makers and third-part suppliers, and can be adapted to different car makers needs. Two complementary points of view have to be specified:

- a set of software components that is represented by a class diagram built from a set of design patterns [5, 18], which specifies the code sequences (methods and attributes) executed to accomplish the middleware's communication services, and
- a set of tasks executing the middleware's

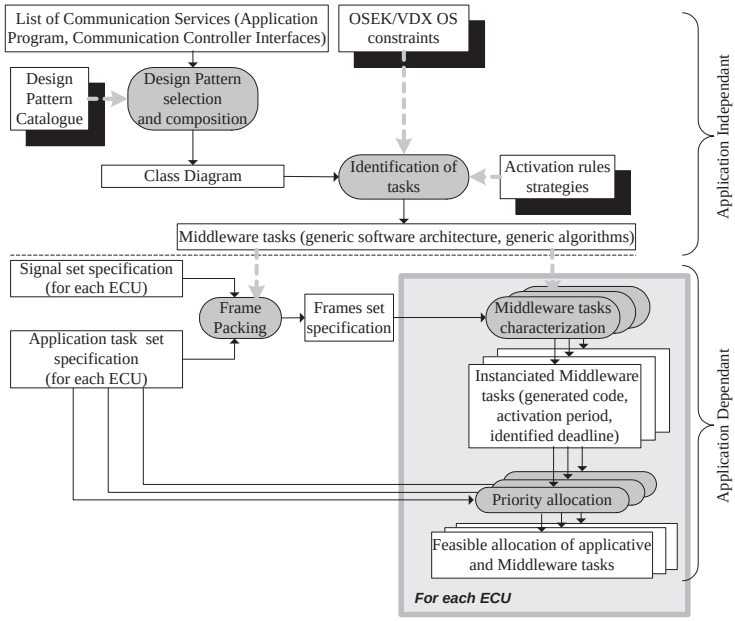

Figure 1. Activities involved in the method for the development of the middleware. Gray ellipses represent the activities. Rectangles illustrate either input/output information (straight arrows) or guiding rules (dashed arrows) for the activities.

communication services, and capable of executing on the OSEK/VDX Operating System (OSEK/VDX OS [15]), which is becoming the standard operating system for event-triggered automotive applications. These tasks, whose code skeleton is generated from the class diagram, are identified using a strategy whose criterion is adapted to the properties of OSEK/VDX OS (limited number of tasks and priorities).

The scheduling parameters of network frames and applicative and middleware tasks are calculated in order to guarantee the respect of the tasks and signals timing constraints. This calculation is performed in two steps:

- the execution of a frame packing algorithm that determines the set of frames transmitted by each ECU over a CAN network [6], as well as, the characteristics of each frame (frame packing configuration) such that the overall bandwidth consumption is minimized, and

- the setting of the characteristics of the middleware tasks (execution time and the activation period) from the frame packing configuration and from the generated tasks source code, and assignment of a feasible priority to each task (applicative and middleware).

The reminder of this paper is organized as follows: section 2 introduces existing studies that are related to the work presented in this paper. Section 3 is devoted to the presentation of middleware's software architecture. In section 4, the approach used to configure the network frames and the tasks is given.

2. Related work

Some initiatives from car makers and third-part suppliers for the definition of a communication middleware exist. OSEK/VDX Communication [14] is one of the modules of an open-ended architecture for distributed electronic components in vehicles. Its goal is to offer a uniform communication interface through an API to transfer data between tasks independently from their location. However, this API is composed of a complex set of services with different semantics, and thus, reducing the portability of the applicative components. Moreover, OSEK/VDX specifies communication services but it does not indicate how to implement them.

Volcano [3] is a commercial in-vehicle communication middleware used by Volvo. It is, on the one

hand, an algorithm of frame packing specifying the set of frames (and their characteristics) exchanged over in-vehicle networks such as CAN [6]. On the other hand, Volcano is a software layer that executes at run-time on each ECU. However, being a commercial product, details on its implementation are not published. Finally, the remaining initiatives are the EAST-EEA [4] and the Autosar projects [2]. The former suffers from the same problem as OSEK/VDX Communication, no hint for implementation, and moreover, there is no publicly available concretization of this architecture. The latter, the Autosar project, tends to create an infrastructure for the management of automotive software modules, such that, the following technical goals are achieved: modularity, scalability, transferability, and reusability. Nevertheless, Autosar is currently under development.

The above mentioned initiatives prove that a firm methodology for the development of an in-vehicle middleware is necessary. Specially, if it decreases the time needed to react to the new demands of automotive functions. For this purpose, the use of design patterns would be an advantage. A design pattern [5, 18] identifies the main aspects of a given object-oriented design structure: the participating classes and objects, their roles, and relations. The goal is to solve design problems arising in a certain context, to make these designs more flexible and reusable, and to improve the documentation and maintenance of existing systems by creating a pattern language. Indeed, design patterns are a good solution to provide portability and interoperability between separately developed software components, which are faced with crucial issues typical of a multitasking context: concurrency and synchronization. To our best knowledge, they have not been yet applied in the automotive systems development, but some work exists concerning their application to the design of the real-time middleware TAO (**T**he **A**CE **O**RB [17]). This middleware, specified using patterns, offers services for applications with real-time QoS requirements. However, it is designed to be dynamically configurable, and due to its resources consumption is not a feasible solution for automotive embedded systems.

3. Middleware software architecture

To present the software components participating in their structure, design patterns use UML class diagrams [12]. This section presents, on the one hand, the class diagram identifying the software components (classes) of the architecture of the middleware, as well as, the design patterns used to achieve it. On the other hand, it introduces the strategy used to identify a set of tasks implementing the software architecture.

3.1. Design patterns for the software architecture

The class diagram representing the software architecture of the middleware is shown in figure 2. It is composed of the set of classes that participate in the used design patterns. Some of these patterns deal with event handling and concurrency issues among objects. Hence, besides establishing the code to be executed, the used patterns identify active objects and thus, influence the middleware's software architecture. In the following, these patterns and their application to the middleware's context are introduced:

- *Adapter* [5]: this pattern allows classes to cooperate together when their interfaces are incompatible. It is composed of an abstract class defining a standard interface to be used by client classes, and of an adapter class that makes the translation between the standard interface and the incompatible one. In figure 2, this pattern is illustrated by a set of adapter classes (*AdMOST* and *AdCAN*), which adjust the interface of in-vehicle networks (MOST [11] and CAN [6] in this case) to a standard set of network services defined in the abstract class *Comm*. This pattern helps the middleware to handle the heterogeneity of communication platforms, and allows the middleware's main class, named *Core*, to be developed and modified independently of the underlying communication network.

- *Observer* [5]: it should be used when an object must notify other objects without making assumptions about which these objects are. This pattern creates a loose dependency between objects, such that, when the state of an object changes all its dependents (or "observers") are immediately notified. It is represented in figure 2, firstly, by classes *Core* and *Comm* that must be immediately notified when a new frame arrives (class *Comm* must notify class *Core*) or is ready to be sent (class *Core* must notify class *Comm*). Secondly, by the abstract class *SubjObs* defining the interface that each "observer" and "observed" class must implement (both classes *Core* and *Comm* are "observer" and "observed"). This pattern permits classes *Core* and *Comm* to evolve independently without hindering the possibility of passing data between them.

- *Asynchronous Completion Token* [18]: the purpose of this pattern is to allow an object to efficiently demultiplex the responses of asynchronous services invoked on other objects. For that, when an asynchronous service is invoked, the invoker passes a token (under the form of an ob-

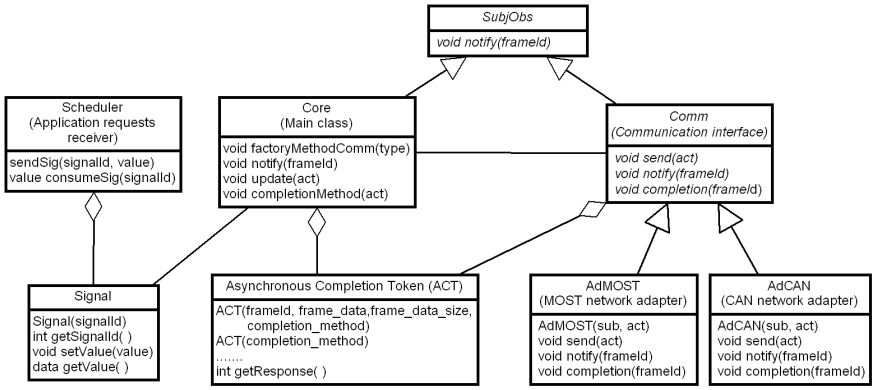

Figure 2. UML class diagram representing the software architecture of the middleware. The classes are the actors of the used design patterns.

ject) containing information that identifies the method that will be responsible for processing the service's response. When the service terminates, the response contains the token and thus, the invoker object can identify the method that will process the response. In the middleware's context, this pattern lets class *Core* (see figure 2) efficiently manage the frame transmission completion events dispatched by the network adapter (class *Comm* in figure 2). If the used communication platform does not provide this type of event, or the service cannot be implemented as asynchronous, the pattern can still be used with the purpose of encapsulating the information exchanged between these two classes. Hence, this pattern contributes to the creation of a loose coupling between middleware classes and still allowing an efficient exchange of data.

- *Integrated Scheduler, variant of the Active Object* [18]: this pattern addresses a concurrency aspect by decoupling the service invocation (occurring in the client's task) from the service execution (happening in a separate task). In the middleware's class diagram of figure 2 the pattern is composed of:

  - a service provider represented by class *Core*,
  - a service requests receiver specified by class *Scheduler*, defining the communication interface provided by the middleware, and
  - a service requests repository depicted by class *Signal*, where applicative tasks store the produced signals and retrieve the signals to consume.

  While class *Scheduler* is executed in applicative tasks, class *Core* is ran in its own task, and class *Signal* represents a shared memory area. Therefore, the functionalities accomplishing the communication services provided by the middleware are executed asynchronously from applicative tasks. These functionalities are, on the one hand, the construction and sending of frames containing the produced signals, and, on the other hand, the reception and handling of the frames carrying the signals to be locally consumed.

The obtained class diagram represents a software architecture independent of real-time requirements. Particularly, it does not contain any reference to the activation rate of the middleware's functionalities. However, this and other timing constraints depend on the demands of the applicative tasks that differ on each ECU. To obtain a representation of the middleware's software architecture that can be easily characterized based on local timing requirements, one can define a set of tasks.

3.2. Middleware tasks identification

For the identification of tasks two assumptions are made. The first is that the frame packing algorithm configures frames with periodic transmission, based on the activation rate of the tasks producing signals. Thus, the functionality in charge of sending frames is activated periodically. The second assumption is that the exact time interval between two consecutive frame arrivals cannot be determined, since a priority bus like

CAN is used. Therefore, the functionality responsible for the receiving frames is triggered sporadically. In this context we consider the following types of functionality activation events:

- *time-triggered*: cyclic timing alarms (supported by the OSEK/VDX OS) for the periodic construction and transmission of frames, and
- *event-triggered*: network controller interrupts indicating the sporadic arrival of frames, and allowing their reception and handling.

To identify a set of middleware tasks we choose the strategy that assigns one task to each different type of functionality activation events. This choice minimizes the amount of middleware tasks, allowing the execution of a maximum number of applicative tasks. Indeed, the specification of the OSEK/VDX OS advises, according to the used conformance class, to limit to 8 or 16 the number of priorities and tasks (the one executing plus those in the ready queue). Otherwise, the portability of the software is not assured.

With the functionality activation events and a sequence diagram, one can determine the chain of methods that is executed by each task. The objective of these methods is, for example, to handle n signals composing a frame received by the network controller. However, n, the number of signals, depends on the frame packing configuration. Consequently, at this stage, one identifies the chain of methods that will be executed but cannot yet generate the complete code of each task, and thus, determine their execution time. In the following section, we detail the procedure used to determine this and the other characteristics of the tasks based on local timing constraints.

4. Configuration of the network frames and the tasks

Since middleware tasks are responsible for the exchange of frames, their characteristics depend on the frame packing (FP) configuration. But, the FP algorithm must be aware of the characteristics of applicative and middleware tasks in order to assign a deadline to each frame that guarantees the signals timing constraints. To overcome this dependency cycle, we propose a three-step algorithm briefly detailed in the next sections.

4.1. Construction of a feasible configuration of frames

The first step of the configuration algorithm is the execution of a FP algorithm such as the *Bi-Directional Frequency Fit* [16] or the *Bandwidth Best Fit decreasing* [8]. The goal is to calculate the FP configuration (signals composing each frame, activation period and deadline of the frames) that minimizes the bandwidth consumption, and respects the timing constraints of signals and frames. The algorithms in [16] and [8] do not consider the delays induced by applicative and middleware tasks. To take these delays into account, and break the dependency cycle, we propose extensions that consider the worst-case permitted behaviour of those tasks. Due to space limitations, the extensions are not presented in this study. The reader can refer to [9] for details. When no feasible configuration is found, the specification of the automotive functions have to be reviewed. Note that this algorithm can also be used with a bottom-up approach, where only a subset of signals are given as input data. The system's designer can then place itself the remaining signals on top of the resulting set of frames.

4.2. Configuration of middleware tasks

The second step consists of setting the parameters of the middleware tasks (execution time, activation period, and relative deadline) on each ECU from the FP configuration. The task in charge of receiving frames is considered sporadic [7], because on CAN the clocks on the different ECUs are unsynchronized, and the exact time interval between two consecutive frame arrivals cannot be determined. The task responsible for sending frames can be implemented as multiframe [10]. This type of task is characterized by a unique activation period and a set of execution times corresponding to successive instances of the task. In our case, this task has different execution times because successive instances transmit a different set of frames (frames may have different transmission periods). The reader can refer to [9] for the procedure used to characterize these tasks from the FP configuration.

4.3. Calculation of a feasible priority allocation

The goal of the third step is to try to find a feasible priority allocation for the set of applicative and middleware tasks of each ECU. In our context, the task receiving frames would be most efficiently implemented as an OSEK/VDX OS interrupt service routine (ISR) triggered by network controller interrupts. Thus, this task would have the highest priority in each ECU because with OSEK/VDX OS, ISRs have necessarily a higher priority than tasks. Moreover, the task sending frames, which is implemented as a classical OS task, would have a higher priority than applicative tasks. Firstly, because communication is the major service on each ECU, an hence, one avoids losing frames due

for instance to buffer overflow. Secondly, middleware tasks can prevent a faulty task from jeopardizing the system's behaviour by consuming all the CPU time due to a software bug for example. The allocation for the entire set of applicative tasks is determined with the optimal Audsley algorithm [1]: if a solution exists then it will necessarily be found. If no feasible priority allocation exists, the specification of the automotive functions have to be re-worked.

5. Conclusion

This study presented a method with the purpose of automatizing the development of the software architecture of an in-vehicle communication middleware, and the calculation of feasible scheduling parameters for network frames and tasks. The software architecture is achieved using design patterns in order to improve the maintenance and the reusability of its components. The scheduling parameters of network frames and tasks are calculated, such that, the respect of the tasks and signals timing constraints is guaranteed.

Future work consists of implementing the software architecture presented in this study, and generating its configuration in conformity with a given set of characterized applicative tasks and signals. The generated configuration must also include the OSEK/VDX Implementation Language [13] files describing the tasks.

References

[1] N. Audsley. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Technical Report YCS164, University of York, November 1991.

[2] AUTOSAR Consortium, 2004. `http://www.autosar.org`.

[3] L. Casparsson, A. Rajnak, K. Tindell, and P. Malmberg. Volcano - a revolution in on-board communications. Technical report, Volvo, 1999.

[4] EAST-EAA ITEA project, 2004. `http://www.east-eea.net`.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] ISO. *ISO 11898 - Road vehicles - Interchange of digital information - Controller Area Network for high-speed Communication*. International Standard Organization, 1994. ISO 11898.

[7] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[8] R. Santos Marques, N. Navet, and F. Simonot-Lion. Frame packing under real-time constraints. In *5th IFAC International Conference on Fieldbus Systems and their Applications - FeT'2003, Aveiro, Portugal*, pages 185–192, July 2003.

[9] R. Santos Marques, N. Navet, and F. Simonot-Lion. Configuration of in-vehicle embedded systems under real-time constraints. To appear in proceedings of the 10th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2005). Available at `http://www.loria.fr/~santos`, 2005.

[10] A. Mok and D. Chen. A multiframe model for real-time tasks. In *RTSS '96: Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS '96)*, page 22. IEEE Computer Society, 1996.

[11] MOST Cooperation. MOST Media Oriented Systems Transport specification, version 2.3, August 2004. Available at `http://www.mostcooperation.com`.

[12] OMG Object Management Group. *OMG Unified Modelling Language: Superstructure*, version 2.0 edition, October 2004.

[13] OSEK Consortium. OIL:OSEK implementation language, version 2.5, July 2004. Available at `http://www.osek-vdx.org`.

[14] OSEK Consortium. OSEK/VDX communication specification, version 3.0.3, July 2004. Available at `http://www.osek-vdx.org`.

[15] OSEK Consortium. OSEK/VDX operating system, version 2.2.3, February 2005. Available at `http://www.osek-vdx.org`.

[16] R. Saket and N. Navet. Frame packing algorithms for automotive applications. Technical Report INRIA RR-4998, 2003.

[17] D. Schmidt and C. Cleeland. Applying patterns to develop extensible ORB middleware. *IEEE Communications Magazine*, 37(4), April 1999.

[18] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.